09/869047
Rec'd PCT/PTO 22 JUN 2001
PCT/CA99/01216

TITLE: METHOD FOR ACCESSING AND RENDERING AN IMAGE

This application is a continuation-in-part of application 08/629,543 filed April 9, 1996.

FIELD OF THE INVENTION

The present invention relates to a method for defining various objects which make up an image and a method of rendering the image on a scanline by scanline basis.

BACKGROUND OF THE INVENTION

There are a number of computer graphics programs which store various objects and use these objects to render the final image. Generally these programs are divided into vector based programs or bitmap based programs. COREL DRAW™ is primarily vector based whereas PHOTOSHOP™ is essentially bitmap based. These known graphics packages allocate enough temporary storage for the entire rendered image and then render each object, one by one, into that temporary storage. This approach fully renders lower objects prior to rendering upper objects. The programs require substantial memory in rendering the final image. Some programs allow an object to be defined as a group of objects and this provides some flexibility. In the case of an object being a group of objects, this group is effectively a duplicate of the base bitmap. Groupings of objects add flexibility in changing the design or returning to an earlier design, but substantial additional memory is required. Changing any object of the image typically requires rerendering of the entire image

The final image of graphics packages is typically sent to a raster device for output, which renders the image on a scanline by scanline basis. The final image

is defined by a host of scanlines, each representing one row of the final bitmap image. Raster devices include printers, computer screens, television screens, etc.

Vector based programs such as COREL DRAW™, produce a bitmap of the final image for the raster device. Similarly, the graphic program PHOTOSHOP™ produces a bitmap of the final image.

Vector based drawings tend to use little storage before rendering, as simple descriptions often produce largely significant results. Vector drawings are usually resolution independent and they are made up of a list of objects, described by a programming language or other symbolic representation. Bitmap images, in contrast, are a rectangular array of pixels wherein each pixel has an associated color or grey level. The bitmap image has a clearly defined resolution (the size of the array). Each horizontal row of pixels of the bitmap is called a scanline. Bitmaps tend to use a great deal of storage, but they are easy to work with because they have few properties.

Recently the definition of bitmaps has been expanded to include an "alpha channel" which represents the transparency of an object or pixel. There are levels of transparency between solid and transparent, which can be represented as a percentage. Although some standard file formats such as CompuServe's GIF are limited to 2 levels (solid and transparent), newer formats such as Aldus' TIFF, PNG (Portable Network Graphics) and the Digital Imaging Group's ".fpx" format allow 256 or more levels of transparency, which allows for smooth blending of layers of content. There remains a need for a method which deals with images in a more consistent manner with respect to bitmap input information and vector based objects.

Our earlier U.S. Patent application SN 08/629,543 entitled Method Rendering an Image allows for scanline based rendering and divides all objects into a tool and region. It is possible to use the region as a local alpha channel for the tool. This doesn't allow for a more general use of alpha channels to create holes in images when used, for example, on web pages -- showing through the background. Also some types of objects such as formatted text with color highlighting cannot be represented easily with a separate region (as the shape of the text), and tool (with the coloring for the text) since particular words need to have different colors, and these need to follow the words when the text is reformatted.

SUMMARY OF THE INVENTION

A method of rendering an image defined by a hierarchy of interacting object according to the present invention comprises

defining each object specifying a region of the image which the object affects, a LookAround distance defining additional input information required by the object to allow the object to output a scanline,

determining for each object rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof, said rendering information comprising said region and said LookAround distance,

defining a hierarchy of said objects which collectively define the image, and

rendering scanlines of said image by determining for each scanline

a) a hierarchy of the active objects which affect the particular scanline of the image,

evaluating the hierarchy of active objects and the rendering information thereof to determine the number of scanlines to be outputted by each active object to render the particular scanline of the image,

using the results of the evaluation of the hierarchy of objects to cause the lowest active object to output the required number of scanlines and pass the scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

According to an aspect of the invention, the method includes the step of defining each object including an alpha channel factor for transparency characteristics of each object.

According to an aspect of the invention, the alpha channel factor of each object is included in said rendering information.

In a further aspect of the invention, the hierarchy of objects includes a background object which is applied as the last active object using said alpha channel factors of said active objects.

In a further aspect of the invention, the lower most active object an input corresponding to a transparent object.

In an additional aspect of the invention at least some of said objects are render layers where each render layer is defined by a separate hierarchy of interacting objects defined in the same manner and each render layer has render layer information corresponding to object rendering information such that said render layers are treated as normal objects relative to higher and lower objects and evaluation of a hierarchy of objects is based on said object rendering information and said render layer information.

In an aspect of the invention a hierarchy of active objects and render layers is used during rendering to form a hierarchy series of alpha channel factors used to determine the transparency associated with each pixel of each scanline.

A draw rendering product of the present invention defines an image as a compilation of hierachial interacting objects; each object being defined by a region of the image which the object affects, a LookAround distance defining scanline input information required by the object to allow the object to output a scanline, and data information of the object. The definition of each object includes rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof. The rendering information comprises the region and the look around distance;

an arrangement for determining on a scanline by scanline basis a hierarchy of the objects which collectively define the image and the required output of each object to return a particular scanline; and

a render engine for rendering a defined image on a scanline by scanline basis which uses the hierarchy of the active objects and the rendering information thereof to determine the number of scanlines to be outputted by each active object to render the particular scanline of the image.

The render engine uses the results of the evaluation of the hierarchy of objects to cause the lowest active object to output the required number of scanlines typically corresponding to only a small portion of the object and pass the scanlines as input to the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

In an aspect of the invention, the draw rendering product includes with each interactive object color and alpha channel information which defines the color and the transparency of the interactive object.

In a further aspect of the invention, the draw rendering product includes defining the lowest object as a transparent background object and defining a desired background as the upper most object.

A draw rendering product according to the present invention defines an image as a compilation of ranked interacting object. Each object is defined by a region of the image which the object affects, a look around factor defining scanline input information required by the object to allow the object to output a scanline, and data information of the object used during rendering of the object. The definition of each object includes rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof. The rendering information comprises the region and the look around factor. The product includes an arrangement for determining object output information for any particular scanline, where the object output information includes a hierarchy of the interactive objects which collectively define the scanline of the image and the required output of each interactive object required by the next highest interactive object to return the particular scanline, and a render engine for rendering a defined image on a scanline by scanline basis which uses the object output information from the lowest to highest interactive objects to cause each interactive object to output the required number of scanlines and pass the scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image. This process is repeated for the next scanline of the image until the entire image has been rendered.

According to an aspect of the invention, the interactive objects include simple objects and render layers where a simple object is defined by a single interactive object and a render layer is defined by its own hierarchy of interactive objects, each render layer including its own render information whereby render information of a render layer allows other objects in a hierarchy with a render layer to use the render information thereof as if the render layer was a simple object.

According to an aspect of the invention, the product includes a rerendering assessment arrangement for rerendering only a portion of a rendered image which has changed due to a change in at least one of said active objects, said assessment arrangement identifying the changed active objects and the scanlines of the rendered image affected by the changed objects and rerendering the required scanlines to replace affected scanlines of the image whereby only a portion of the scanlines of the image are rerendered.

BRIEF DESCRIPTION OF THE DRAWINGS

Preferred embodiments of the invention are shown in the drawings, wherein:

Figure 1 illustrates the Abstract Render Object interface;

Figure 2 illustrates the Abstract Compositing Operator interface;

Figure 3 illustrates the concrete Render Layer Class;

Figure 4 displays the inter-relationship of the various major classes used to define the method; and

Figures 5 through 8 illustrate different effects by combining objects in different layers;

Figures 9 and 10 illustrate a series of RenderObjects and the steps for returning 3 scanlines; and

Figures 11, 12 and 13 show the structure of an image, the actual image (Fig. 12) and components of the layer of the image (Fig.13).

Appendix A contains pseudo code illustrating the steps involved in rendering an image to an output device.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENTS

Figures 1, 2 and 3 explain various features of the present invention regarding the approach used to produce an image defined by a series of elements of the image.

Fundamental to the present invention is the Abstract RenderObject interface. The Abstract RenderObject interface 2 of Figure 1 is a very broad flexible definition. This definition includes a RenderObject 11, which expects to receive color information 4 and alpha information 6 as an input and will produce color information 8 and alpha information 10 as an output. The definition expects to receive what is referred to as the BoundBox 12 of the object which is typically a two-dimensional rectangle as well as a factor referred to as LookAroundDistance 14. The LookAroundDistance 14 is the additional information required by the object over and above the BoundBox 12 which is required by the object to render itself. For example, if the RenderObject 2 included a blur tool, it would require some additional information exterior to the BoundBox to effect rendering. This additional distance is defined by the LookAroundDistance. Any object that meets this criteria is acceptable to the Abstract RenderObject interface. With this broad definition, different types of objects can easily be defined and all objects interact in a common manner. A series of objects having this definition are ranked from the deepest object to the shallowmost object and the partial or full output from each object is provided to the appropriate upper object. Basically, this is similar to a vector based program. However, this definition can also

cover other objects such as a bitmap. In this way, a host of different type of RenderObjects can cooperate to effect the final image.

This definition and approach allows simplified rendering where only the objects and the portions thereof affecting a scanline are considered to render that scanline.

Figure 2 shows an Abstract CompositingOperator Interface 20. This interface again is looking for color and alpha information regarding the background 4a, 6a. Color and alpha information (4b,6b) regarding the foreground and it will output color and alpha information 4c, 6c.

The Abstract CompositingOperator Interface 20 of Figure 2 is illustrated in association with the Concrete RenderLayer Class 30 of Figure 3. The Concrete RenderLayer Class, to RenderObjects above or below this class acts and looks like, a simple RenderObject. The Concrete RenderLayer Class 30 requires color and alpha information 32, 34 as an input and will output color and alpha information 36, 38. The Concrete RenderLayer Class differs from the RenderObject 11 of Figure 1 in that it contains ranked RenderObjects 11 and combines the objects in a sequential manner by feeding the required scanline output of the object as an input to the next object. The transparent background 39 is introduced as initial color and alpha information to the first RenderObject. The input 32, 34 provided to the Concrete RenderLayer Class is combined with the output of the last ranked object within the layered class using the CompositingOperator 20. This will result in the desired outputted color and alpha information 36, 38. In this way, an entire layer is affectively represented within a series of objects which are collectively defined by a single series of ranked Objects and RenderLayers. With this system very

sophisticated images can be defined and the image is easily changed. It can also be appreciated that the Concrete RenderLayer Class of Figure 3 can itself include, as one of the objects, a further Concrete RenderLayer Class. This nesting approach can be repeated as necessary to define the particular image.

The Concrete RenderLayer Class of Figure 3 also has, as part of its definition, the overall BoundBox and LookAroundDistance associated with the collective definition of the Render Layer. In this way, an image is affectively represented as a single series of ranked objects and RenderLayers and it is immediately apparent, based on the BoundBox and the LookAroundDistance, which objects affect particular lines of the image. This is important as the present invention examines the list of objects and layers when it is about to return a scanline to determine which objects affect that particular scanline and then provide the proper series of objects for returning of that scanline. In addition, a determination is made with respect to how many scanlines each object or layer must return to return the desired scanline such that each object in the series will produce sufficient lines to cause the uppermost object or layer to return the desired scanline. This results in a one pass approach and provides efficiencies with respect to rendering of the image.

The fact that a Concrete RenderLayer has a BoundBox and LookAroundDistance as all other RenderObjects allows a single series of "RenderObjects" (objects + render layers) to be assessed to determine the partial object outputs to return the desired scanline.

Figure 4 is an overview of the arrangement and illustrates how different elements of an image can be defined as a Render Object. As shown, a RenderLayer is a type of RenderObject. An effect which includes a Region and Tool type definition is a RenderObject and the program

also accommodates any other RenderObject which meets the definition of a RenderObject to be included. These include Bitmaps with alpha channel information and specialty text type features. It should be noted that under the Tool Object there is a LookAround Tool which requires several scanlines to allow rendering. It can be seen that the Blur Tool is one example of a LookAround Tool. Although the RenderLayer has been shown as a type of RenderObject, the RenderLayer itself may have RenderLayers within it. This aspect will be more fully explained with respect to Figures 5 through 8.

Figure 5a shows a color fade background 200 having a black rectangle 202 placed thereon with a blue opaque circle 204 thereabove with a blue opaque triangle 206 located above the circle 204. Figure 5b shows the series of RenderObjects used to produce the image 208. A transparent background 199 is fed to the RenderObject which defines the color fade background 200 which then feeds the color and alpha information to the RenderObject defined by the black rectangle 202 which passes the output to the RenderObject 204 defining the blue opaque circle which then feeds the output to the RenderObject 206 which effectively finishes the image. Such a series of objects can be determined for each particular scanline using the BoundBox and LookAround factor, and some of these objects will drop out depending upon the particular scanline. In any event, it can be seen that these RenderObjects form a single series, one above the other and produce the image of Figure 5a.

Figure 6a includes the concrete RenderLayer 212 which has been combined with the color fade background 200. The concrete RenderLayer is defined by the black opaque rectangle 202a. A punch out of the opaque circle 204a and a punch out of the opaque triangle 206a. The punch out capabilities of the RenderObjects 204a and 206a are relatively straight forward due to the alpha channel

information. The punch out feature punches out all underlying Objects. The background is exposed within the punchout as the background was not part of the Render Layer. The series of RenderObjects 202a, 204a and 206a have the results thereof fed to the compositor 214 which then combines this onto the color fade background 200. As both the circle and the triangle are opaque the punch out fully exposes the background in the punch out.

In Figure 7a, the color fade background 200 is combined with a Concrete object layer 220 which is defined by a black rectangle 202b, a punch out translucent circle 204b and a punch out translucent triangle 206b. This result is then combined by the compositor 222 which places the results of the Concrete layer on top of the color fade background 200. In this arrangement, the overlap of the circle and triangle produce a pie-shaped lighter region and the background within the keyhole is darker than in Figure 6a as the punch out was only of translucent circle and triangle. The lighter pie shaped region is due to the overlapping of the translucent circle and translucent triangle.

In Figure 8a, a more realistic keyhole is defined. This drawing is the result of a color fade background 200, a first Concrete layer 226 having an additional Concrete layer 228 PunchOut layer 228 is combined with a black rectangle in layer 226. Concrete Layer 226 receives the colorfade background as an input and outputs the image.

From a planning point of view, the drawing of Figure 8a at the uppermost level is a simple arrangement having a color fade background and a Concrete layer 226. The concrete layer 226 has the required BoundBox and LookAroundDistance associated with the layer and basically appears to be a simple RenderObject. Any planning with respect to the number of scan lines of input required by each of the RenderObjects or RenderLayers to return a scan

line is easy to determine. This same upper level approach is found with respect to each of the Concrete layers.

The present program uses the uppermost series of RenderObjects and RenderLayers to determine the requirements from top down of each of the objects or layers and treats a Concrete RenderLayer as a RenderObject. For example, you can look at the uppermost object and for a particular scanline, the number of required input scanlines for the Object is effectively defined by the LookAroundDistance. The next object knows the output required of it by higher objects and then determines how many lines it will require etc., down to the lowermost RenderObject. In this way, the requirements of each RenderObject are known and each RenderObject will produce the desired number of scanlines from lowest to highest based on the earlier assessment. This guarantees that there will be an output scanline for each trip through the RenderObjects and RenderLayers. Only the portions of each object necessary for rendering the desired scanline is evaluated. This approach also allows effective temporary storage of certain information required of the concrete layers during the rendering thereof. These can be stored in suitable buffer memories.

Figure 9 shows a composition of three render objects and will be briefly explained with respect to what might be considered scanlines 1, 2, and 3 whereas in actual fact these are somewhat larger areas. The objects defining the image of Figure 9 are "Heart1" which displays a red heart 310, "Hello1" which displays the black text "Hello World" 312, and "Blur1" 314 which blurs or makes fuzzy the content underneath it. Heart1 and Hello1 are known as "simple render objects" because they require as input only the background immediately behind each output pixel. This information can be ascertained by using the getLookAroundDistances for the given object. For a simple object, the LookAround distance is 0. This call passes in

the output resolution and a transformation to the output space (which can involve rotation, scaling, translation and skewing - ie. all affine transformations). The result is the number of extra pixels required as input which are above, below, to the left and to the right of any given pixel, in order for the object to be rendered. When the number of extra pixels is 0 in every direction, the object is considered to be a simple object. If the number is greater than zero in any direction then the object is a "LookAround" object. An example of a LookAround object is Blur1.

Blur1 requires an extra pixel in each direction to render its effect. The extra area required by the blur is shown by the dashed line 316 around the blur's BoundBox 315 in Figure 3. Note that the blur requires information below the third scanline, which means that an additional scanline which isn't output needs to be at least partially rendered.

Using the rendering method, shown in Appendix A and the ranking of objects and layers of Figure 10, the render engine is invoked on the containing RenderLayer, called "RenderLayer1." RenderLayer1 returns a render job object identified here as "RenderLayerJob1." To get a scanline, the renderScanline method is called on RenderLayerJob1, passing in a background. RenderLayerJob1 determines which objects affect the Scanline 1 and renders them completely (Figure 10 steps 1 and 2). The result step 2 is needed by the blur, which is buffered for later use. The resulting Scanline 1 is then returned at step 3. The next time renderScanline is called (i.e. for scanline 2), the blur becomes active. Since the blur needs a pixel above and a pixel below it as input in order to render correctly, the RenderLayerJob1 must buffer up more information. The result of steps 4, 5 and 6 is buffered as well as the results of steps 7 and 8. These three results (from step 2, 6 and 8) are then passed into the BlurJob1

which results in step 9. The buffer from step 2 can now be discarded or marked for reuse. The resulting scanline 2 is returned at step 10. To rendered scanline 3, the blur requires more than the already buffered result of step 6 and step 8, and so RenderLayerJob1 renders step 11 and step 12. These three buffers (from step 6, 8 and 12) are then passed into the BlurJob1 which results in step 13. Finally the scanline 3 is returned in step 14, and all of the temporary buffers can be discarded.

In this example, only 3 scanline buffers were required versus 4 scanlines buffers which would have been required with a Painter's Algorithm. With a larger render, the resource savings are often significant. Also the result of the top of the image (initial scanlines) became available much earlier.

Figures 1, 11 and 12 show an example of the processing of images by the modules in the invention. Figure 11 shows the single series Heart1, Layer 1 and Bitmap 1 which produce the image of Figure 12. Heart1 and Bitmap 1 are simple RenderObjects. Layer 1 is defined by series Wave 1 Shadow 1 and Text 1. The results of Layer 1 are shown in Figure 13. The Wave 1 is confined within Layer 1 and does not wave the underlying Bitmap 1 and does not wave the single object Heart1 which receives the output of Layer 1.

Figure 12 shows a composite image comprising a heart (Heart 1), stylized text "Exploring the Wilderness" (Layer 1) and a bitmap image of an outdoor scene (Bitmap 1) underneath the heart and the stylized text. The stylized text is shown with its normal attributes at b, with a shadow at c and with a wave at d.

As shown in figure 9, the invention processes each element of the image according to a hierarchical stack, having the heart ("Heart1") at the top of the stack,

the stylized text ("Layer1") in the next layer down and finally with the bitmap ("Bitmap1") at the bottom. Layer1 is exploded to show its constituent effects, comprising a wave effect ("Wave1"), a shadow effect ("Shadow1") and the text ("Text1").

The general structure of Figure 11 isolates the dependencies between parent and child elements to one level of abstraction. As such, the invention provides abstraction between and amongst elements in an image. This abstraction provides implementation efficiencies in code re-use and maintenance. It can be appreciated that for more complex images having many more elements, bitmaps and effects, the flexibility and efficiencies of using the same code components to processes the components of the image become more apparent.

In the preferred embodiment, exactly one scanline is rendered during each call to the render method on any render object. This even holds for Render Layers, since a Render Layer constitutes a valid implementation of the Render Object class. In the example implementation, Render Layer always passes a completely transparent background as input to its bottommost object. Then the scanlines produced by applying the bottom most object to the transparent background scanlines are passed as input to the next highest object. Similarly, the output of the second object is passed as input to the third object from the bottom. This passing repeats until the cumulative effect of all of the render group's objects is produced. The final results are then composited onto the background scanlines (passed by the caller) using the render group's compositing operator. A Render Layer produces the number of scanlines required by higher ranked Objects or Layers to return the desired scanline.

Because some render objects have forward look-around, it is often necessary for lower objects to render a

few scanlines ahead of objects above them. For example, for an object with one scanline of forward look-around to render a single scanline within its active range, the object immediately below it must already have rendered its result both on that scanline and on the following scanline. Since rendering is performed from the bottommost object to the topmost object, the process is begun by determining exactly how many scanlines must be rendered by each object and/or Render Layer.

The computation is most easily done in terms of the total number of scanlines rendered by each object so far during the entire rendering process, as opposed to the number of scanlines rendered by each object just during this pass. The total number of scanlines required _of_ an object is referred to, relative to that object, as downTo whereas the total number of scanlines required _by_ an object is referred to, relative to that object, as downToNeeded. Note that the downToNeeded of a given object is always equal to the downTo of the object immediately below it, if applicable. In the case of the bottommost object, its downToNeeded is the number of empty input scanlines that must be passed to it in order for it to satisfy the object above it, if any, or the caller otherwise.

The present arrangement allows a complicated compilation of different objects and layers to be treated in a common manner and to allow assessment of output information to be determined on a single ranking. This assessment is repeated for each Render Layer when the output of the Render Layer is required. The assessment does not require the full evaluation of the Render Layer as it is treated as a Render Object for assessment purposes. This common approach allows simplification in processing and efficiencies in processing.

Although various preferred embodiments of the present invention have been described herein in detail, it will be appreciated by those skilled in the art, that variations may be made thereto without departing from the spirit of the invention or the scope of the appended claims.

APPENDIX A

Abstract Interfaces for Hierarchical Model

```
Interface RenderObject
        Rect2D getBoundBox(Affine2D objToHome)
        LookAroundDistances getLookAround(int width, int
height, Affine2D objToHome)
        RenderJob initRender(int width, int height,
Affine2D objToHome)
EndInterface
```

Interface RenderJob

```
        boolean prefersToOverwriteInputScanline()
        LookAroundDistances getLookAround(int width, int
height,Affine2D objToHome)
        void renderScanline(RGBAScanline inputBuffer,
RGBAScanline outputBuffer)
EndInterface
```

Support Classes.

```
Class RGBAScanline
        Constructor(int length, LookAroundDistances
look)
    Byte[] getRGBData(int scanline)
    Byte[] getAlpha(int scanline, int offset)
    Int getRGBOffset(int scanline)
    Int getAlphaOffset(int scanline)
EndClass

Class LookAroundDistances
        Int left,right,up,down
EndClass

Class Rectangle2D
        Double x,y,width,height
EndClass

Class Affine2D
        [ a11 a12 a13]
        [ a21 a22 a23]
        [   0     0     1 ]
EndClass
```

RenderLayer and RenderLayerJob Psudocode

This algorithm is presented here in simplified form in order to clearly convey its fundamental concepts. All trivial modifications or extensions to the algorithm as presented here should be considered as such, even if the modifications are only conceptually trivial, though complex in implementation.

The % operator is defined as follows: a % b := b*(a/b-floor(a/b))

```
-----------------------------------------------------------------------
-

// Each instance of this class represents an active render process
// for a render object.  Rendering begins by calling
// RenderObject.initRender(), which returns a render job.
abstract class RenderJob
{
        // Renders a single scanline of an object onto the given
        // scanline, storing the result in the given output buffer.
            It
        // is acceptable for out to actually be the central scanline
            of
        // in.  The caller is responsible for making sure that in
        // contains a sufficient amount of look-around information.
        void renderScanline( LookAroundScanline in, PaddedScanline
out);
}

// This class represents a 2D object which has some appearance or
// effect when rendered onto a background.
abstract class RenderObject
{
        // Returns the maximum required look-around for rendering the
given
        // rectangle.
        LookAround getLookAround( Rectangle r);

        // This method should return true if this object's render
jobs
        // would prefer to write to the same buffer from which they
read.
        // This allows for a simple but effective optimization.
        boolean prefersToOverwriteInputScanline();

        // Returns a 'coverage' object loosely describing the maximum
        // extent of this object's visible effect, regardless of the
        // appearance of the background.  See also: Coverage.
        Coverage getDomain();

        // Constructs a new render job for this object which will
render
        // all of the pixels in the given rectangle.
        RenderJob initRender( Rectangle r);
}
```

```
abstract class CompositingOperator
{
        // Composits the two given colours, and returns the resulting
        // colour. Note that the term `colour' is used loosely here,
        // and includes an alpha component.
        RGBA composit( RGBA background, RGBA foreground);
}

abstract class Coverage
{
        // Returns a rectangle that completely contains the area
        // represented by this object.
        Rectangle getBoundbox();
}

// This class represents a horizontal line of RGBA values, as a sub-
// array of an array of RGBA values.  This is a valuable alternative
to
// just a plain array, since it allows for look-around pixels and
// sub-scanlines.
class PaddedScanline
{
        private int offset;
        private int width;
        private RGBA[] buffer;

        // Constructs a padded scanline with a specified amount of
        // padding to the left and right.
        PaddedScanline( int width, int lookLeft, int lookRight)
        {
                offset = lookLeft;
                this.width = width;
                buffer = new RGBA[ width + lookLeft + lookRight];
        }

        // Construct a sub-scanline of the given scanline.
        private PaddedScanline( PaddedScanline s, int offset,
            int width)
        {
                this.offset = s.offset + offset;
                this.width = width;
                buffer = s.buffer;
        }

        // Returns a sub-scanline of this scanline.
        PaddedScanline subScanline( int offset, int width)
        {
                return new PaddedScanline( this, offset, width);
        }
```

```
        // Gets the colour and alpha of the pixel at a specified
offset
        // from the left edge of this scanline, not counting any
        // additional padding.
        RGBA getPixel( int x)
        {
                return buffer[ x + offset];
        }

        // Sets the colour and alpha of the pixel at a specified
offset
        // from the left edge of this scanline, not counting any
        // additional padding.
        void setPixel( int x, RGBA colour)
        {
                buffer[ x + offset] = colour;
        }

        // Copies the contents of this scanline, plus additional
        // look-around information to the left and right, into the
        // given scanline.
        void blitTo( PaddedScanline dest, int lookLeft, int
lookRight)
        {
                // If the source and dest scanlines are the same
                // scanline, then we can just return without doing
                // anything.
                if( this == dest) return;

                for x = -lookLeft to (width+lookRight-1)...
                        dest.buffer[ x + dest.offset] = buffer[ x +
                        offset];
        }

        // Composits the contents of this scanline, plus additional
        // look-around information to the left and right, into the
        // given scanline. The given compositing operator is used to
        // perform the compositing.
        void mixTo( PaddedScanline dest, CompositingOperator
operator,
                int lookLeft, int lookRight)
        {
                for x = -lookLeft to (width+lookRight-1)...
                {
                        dest.buffer[ x + dest.offset] =
                              operator.composit(
                              dest.buffer[ x + dest.offset],
                              buffer[ x + offset]);
                }
        }
}
```

```
// Represents a scanline with additional look-around in all four
// directions.
class LookAroundScanline
{
        private int offset;
        private PaddedScanline[] buffer;

        // Constructs a look-around scanline with the specified
amount
        // of vertical look-around.  This constructor does not
      allocate
        // the scanlines themselves.  Therefore, the object will not
      be
        // complete until all of the scanlines have been set using
      the
        // setScanline() method.
        LookAroundScanline( int lookUp, int lookDown)
        {
                buffer = new PaddedScanline[ 1 + lookUp + lookDown];
                offset = lookUp;
        }

        // Returns a specified padded scanline.  A y value of 0
returns
        // the central scanline.  Negative values return higher
        // scanlines.  Positive values return lower scanlines.
        PaddedScanline getScanline( int y)
        {
                return buffer[ offset + y];
        }

        // Sets the scanline at the given y position.  0 is the
central
        // scanline.  Negative values may be used to set higher
        // scanlines.
        void setScanline( int y, PaddedScanline s)
        {
                buffer[ offset + y] = s;
        }

        // Returns the colour and alpha values of the pixel at the
        // given coordinates in this scanline.
        RGBA getPixel( int x, int y)
        {
                return getScanline( y).getPixel( x);
        }

        // Sets the colour and alpha values of the pixel at the given
        // coordinates in this scanline.
        void setPixel( int x, int y, RGBA colour)
        {
                getScanline( y).setPixel( s, colour);
        }
}
```

```
// This is a special render object that contains a list of render
// objects.  The effect of rendering this object onto a given
// background is the same as rendering each of its contained objects
// one by one onto a blank background, then compositing the resulting
// image onto the given background.
class RenderLayer extends RenderObject
{
        Vector objects;
        CompositingOperator op;

        LookAround getLookAround( Rectangle r)
        {
                return {0,0,0,0};
        }

        boolean prefersToOverwriteInputScanline()
        {
                return true;
        }

        // Returns the union of the domains of the contained objects.
        Coverage getDomain()
        {
                Coverage r = empty;

                for each contained object...
                        r = Union( r, object.getDomain());

                return r;
        }

        // Initializes the render job.
        RenderJob initRender( Rectangle r)
        {
                return new RenderLayerJob( this, r)
        }
}
```

```
// For use by RenderLayerJob, this class is essentially a wrapper
// around RenderJob which avoids initialization until it is
absolutely
// necesary, then automatically goes away when it is done.  It also
// collaborates with the RenderLayerJob in order to share buffers
// effectively with other job nodes.
class JobNode
{
        RenderObject object;
        Rectangle rect;
        LookAround look;
        int nextOut;
        RenderJob job;
        PaddedScanline[] backBufs;
        private LookAroundScanline rgbaScanline;
        int downTo;

        JobNode( RenderObject object, Rectangle r)
        {
                this.object = object;
                rect = object.getDomain().getBoundbox();
                look = object.getLookAround( r);
                nextOut = rect.top - look.up;
        }

        // Computes how many background scanlines must be passed
before
        // a total of downTo scanlines.
        int downToNeeded()
        {
                if( downTo < rect.top)
                {
                        return downTo;
                }

                if( downTo < rect.bottom)
                {
                        return downTo + look.down;
                }

                if( downTo < rect.bottom + look.down)
                {
                        return rect.bottom + look.down;
                }

                return downTo;
        }
```

```
// Renders a single scanline of this object.  For efficiency,
// rendering is restricted both horizontally and vertically
// to
// pixels contained in this object's affected area.
// Vertically, this restriction is accomplished by deferring
// initialization of the render job until the object's first
// scanline is reached, then by throwing away the render job
// when its last scanline is reached.  Horizontally, the
// restriction is accomplished by passing sub-scanlines of
// the
// input buffers to the render job.
boolean renderScanline()
{
        // If we haven't yet reached scanline rect.top-
        // look.up
        // then there is nothing to do yet.
        if( nextOut < rect.top - look.up) return false;

        // Once we reach the scanline a few scanlines above
        // this object's bound box, we need to begin
        // buffering
        // background scanlines.
        // This happens look.up scanlines above the object.
        if( nextOut == rect.top - look.up)
        {
                // Create the back buffers for look-around
                // objects
                // Note that non-look-around objects that
                // don't
                // overwrite their input buffer also need a
                // back buffer.

                if( look.top > 0 ||
                      !job.prefersToOverwriteInputScanline())
                {
                    backBufs = new PaddedScanline[
                        look.up + 1];

                    for i = 0 to look.up...
                          backBufs[i] = new
                            PaddedScanline(
                            rect.width, look.left,
                            look.right);
                }
        }
```

```
// If we've reached the object's first scanline,
// construct its render job.
if( nextOut == rect.top)
{
        job = object.initRender( rect);

        // We also need to construct a look-around
        // scanline as
        // input to the renderScanline() method.
        rgbaScanline = new LookAroundScanline(
              rect.width,
              look.up, look.down);
}

// The global buffers are where the final results are
// built up. The following line decides which global
// buffer this object's current scanline will be
// rendered into.  At this point, this same global
// buffer is guaranteed to contain the appropriate
// final results of all underlying objects.

// Compute which global buffer represents the current
// scanline.
PaddedScanline globalBuf = globalBufs[ nextOut %
  numGlobalBufs].subScanline( rect.left);

// If we have not yet reached the object, all we need
// to do is buffer the current scanline for next
// pass.
if( nextOut < rect.top)
{
        globalBuf.blitTo( backBufs[ nextOut %
              (look.up + 1)],
              look.left, look.right);

        nextOut ++;

        return false;
}
```

```
// Set the look-down information to the contents of
//  the
// appropriate global buffers.  Note that for look-
// down
// information, we are guaranteed that the contents
 // of
// the global buffers has not yet been overidden.  /
// This
// is not true for look-up information.  This is why
 // we
// need to keep back-buffers, but not front-buffers.
for i = 0 to look.down...
{
        rgbaScanline.setScanline( i,
                globalBufs[ (nextOut + i) %
                    numGlobalBufs].
                    subScanline( rect.left));
}

// Compute the back buffer corresponding to the
// current
// scanline.
PaddedScanline backBuf = backBufs[ nextOut %
  (look.up + 1)];

// If this job prefers not to overwrite its input
// scanline, then let's give it the current back
// buffer
// as input. (The current back buffer will shortly
// become a copy of the current global buffer.)
if( !job.prefersToOverwriteInputScanline)
{
        rgbaScanline.setScanline( 0, backBuf);
}

if( look.up > 0 ||
  !job.prefersToOverwriteInputScanline())
{
        // Make a local copy of the input scanline
        // that
        // we're about to overwrite
        globalBuf.blitTo( backBuf, look.left,
        // look.right);
```

```
                        // Set all of the look-up scanlines to
                        // previously stored back buffers.
                        int backBufNum = nextOut;
                        for i = -look.up to -1...
                        {
                                backBufNum ++;
                                rgbaScanline.setScanline( i,
                                        backBufs[ backBufNum %
                                        (look.up + 1)]);
                        }
                }

                // Render the scanline
                job.renderScanline( rgbaScanline, globalBuf);

                nextOut ++;

                // Return true if and only if we just rendered the
                // object's final scanline.
                return (nextOut == rect.top + rect.height);
        }
}

PaddedScanline[] globalBufs;
int numGlobalBufs;

// This is the implementation of RenderJob for render groups.
class RenderLayerJob extends RenderJob
{
        NodeList list;
        Rectangle rect;
        LookAround look;
        PaddedScanline tmpScanline1;
        PaddedScanline tmpScanline2;
```

```
RenderLayerJob( RenderLayer l, Rectangle r)
{
        rect = r;

        int maxLookDown = 0;

        // Construct a job node for each object being
        // rendered.
        Rectangle clipRect = r;
        for each object in l.objects from last to first...
        {
                JobNode node = new JobNode( object, r);

                // Compute the object boundbox, expanded by
                // the
                // look-around distances.
                Rectangle eRect;
                eRect.left = node.rect.left - node.look.left;
                eRect.top = node.rect.top - node.look.up;
                eRect.right = node.rect.right +
                      node.look.right;
                eRect.bottom = node.rect.bottom +
                      node.look.down;

                clipRect = Union( clipRect, eRect);

                maxLookDown += node.look.down;

                list.add( node);
        }

        // Compute the required number of global buffers.
        // This
        // calculation could be improved significantly.  This
        // number must be at least 1 + max( look0,
    maxLookDown)
        // where look0 is the total vertical look-around at
        // scanline 0, and maxLookDown is the maximum
        // downwards look-around for any scanline of the
        // image.
        // It is easy to show that the number computed here
        // is
        // sufficient.
        numGlobalBufs = 1 + (rect.top - clipRect.top) +
          maxLookDown;

        // Allocate the global buffers.
        globalBufs = new PaddedScanline[ numGlobalBufs];
        for i = 0 to numGlobalBufs-1...
                      globalBufs[ i] = new PaddedScanline(
                        rect.width,
                        rect.left - clipRect.left,
                        clipRect.right - rect.right);

        nextOutLine = 0;
        nextInLine = clipRect.top;
}
```

```
void renderScanline( LookAroundScanline in, PaddedScanline
out)
{
        int downto = nextOutLine;

        // Compute how many scanlines are needed by each
        // object
        for each node in list...
        {
                node.downTo = downTo;
                downTo = node.downToNeeded();
        }

        // Empty out the background
        while( nextInLine <= downTo)
        {
                globalBufs[ nextInLine % numGlobalBufs].fill(
                        RGBA.CLEAR);

                nextInLine ++;
        }

        // Render necessary scanlines
        for each node in list backwards...
        {
                while( node.nextOutLine <= node.downTo)
                {
                        boolean done = node.renderScanline();
                        if( done)
                        {
                                list.remove( node);
                                break;
                        }
                }
        }

        // Composit the result with the background
information.
        in.getScanline( 0).blitTo( out, 0, 0);
        globalBufs[ nextOutLine % numGlobalBufs].
                mixTo( out, operator, 0, 0);

        nextOutLine ++;
    }
}
```

THE EMBODIMENTS OF THE INVENTION IN WHICH AN EXCLUSIVE PROPERTY OR PRIVILEGE IS CLAIMED ARE DEFINED AS FOLLOWS:

1. A method of rendering an image defined by a hierarchy of interacting objects, said method comprising

defining each object specifying a region of the image which the object affects, a LookAround distance defining additional input information required by the object to allow the object to output a scanline,

determining for each object rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof, said rendering information comprising said region and said LookAround distance,

defining a hierarchy of said objects which collectively define the image, and

rendering scanlines of said image by determining for each scanline

a) a hierarchy of the active objects which affect the particular scanline of the image,

evaluating the hierarchy of active objects and the rendering information thereof to determine the number of scanlines to be outputted by each active object to render the particular scanline of the image,

using the results of the evaluation of the hierarchy of objects to cause the lowest active object to output the required number of scanlines and pass the scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

2. A method as claimed in claim 1 wherein said step of defining each object includes defining an alpha channel factor for transparency characteristics of each object.

3. A method as claimed in claim 2 wherein said alpha channel factor of each object is included in said rendering information.

4. A method as claimed in claim 3 wherein said hierarchy of objects includes a background object which is applied as the last active object using said alpha channel factors of said active objects.

5. A method as claimed in claim 3 including providing to said lower most active object an input corresponding to a transparent object.

6. A method as claimed in claim 4 wherein at least some of said objects are render layers where each render layer is defined by a separate hierarchy of interacting objects defined in the same manner and each render layer has render layer information corresponding to object rendering information such that said render layers are treated as normal objects relative to higher and lower objects and evaluation of a hierarchy of objects is based on said object rendering information and said render layer information.

7. A method as claimed in claim 6 wherein some of said render layers are defined as a hierarchy of objects and at least one render layer.

8. A method as claimed in claim 7 wherein said hierarchy of active objects and render layers is used during rendering to form a hierarchy series of alpha channel factors used to determine the transparency associated with each pixel of each scanline.

9. A draw rendering product for defining an image as a compilation of hierachial interacting objects; each object being defined by a region of the image which the

object affects, a LookAround distance defining scanline input information required by the object to allow the object to output a scanline, and data information of the object which includes defining the data information using vector based techniques or bitmap techniques;

the definition of each object including rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof, said rendering information comprising said region and said look around distance,

an arrangement for determining on a scanline by scanline basis a hierarchy of said objects which collectively define the image and the required output of each object to return a particular scanline,

a render engine for rendering a defined image on a scanline by scanline basis which uses said hierarchy of the active objects and the rendering information thereof to determine the number of scanlines to be outputted by each active object to render the particular scanline of the image,

said render engine using the results of the evaluation of the hierarchy of objects to cause the lowest active object to output the required number of scanlines typically corresponding to only a small portion of the object and pass the scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

10. A draw rendering product as claimed in claim 9 including with each interactive object color and alpha channel information which defines the color and the transparency of the interactive object.

11. A vector based draw rendering product as claimed in claim 10 including defining the lowest object as a

transparent background object and defining a desired background as the upper most object.

12. A draw rendering product for defining an image as a compilation of hierachial interacting objects; each object being defined by a region of the image which the object affects, a look around factor defining scanline input information required by the object to allow the object to output a scanline, and data information of the object used during rendering of the object;

the definition of each object including rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof, said rendering information comprising said region and said look around factor,

an arrangement for determining object output information for any particular scanline, said object output information including a hierarchy of said interactive objects which collectively define the scanline of the image and the required output of each interactive object required by the next highest interactive object to return the particular scanline,

a render engine for rendering a defined image on a scanline by scanline basis which uses said object output information from the lowest to highest interactive objects to cause each interactive object to output the required number of scanlines and pass the scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

13. A product as claimed in claim 12 wherein said interactive objects include simple objects and render objects where a simple object is defined by a single interactive object and a render object is defined by its own hierarchy of interactive objects, each render object

including its own render information whereby render information of a render object allows other objects in a hierarchy with a render object to use the render information thereof as if the render object was a simple object.

14. A product as claimed in claim 13 including a rerendering assessment arrangement for rerendering only a portion of a rendered image which has changed due to a change in at least one of said active objects, said assessment arrangement identifying the changed active objects and the scanlines of the rendered image affected by the changed objects and rerendering the required scanlines to replace affected scanlines of the image whereby only a portion of the scanlines of the image are rerendered.

15. A product as claimed in claim 14 wherein said rendering information of each object includes an alpha channel factor which defines transparency characteristics of each object.

16. A draw rendering product for defining an image as a compilation of hierachial interacting objects; each object being defined by a region of the image which the object affects, a look around factor defining scanline input information required by the object to allow the object to output a scanline, and data information of the object defined using vector based techniques or bitmap techniques;

the definition of each object including rendering information which allows assessment of the interaction of the particular object with other objects without rendering thereof, said rendering information comprising said region and said look around factor,

an arrangement for determining object output information for any particular scanline or group of scanlines, said object output information including a hierarchy of said interactive objects which collectively define the scanline of the image and the required output of

each interactive object required by the next highest interactive object to return the particular scanline or group of scanlines..

17. A render engine for rendering a defined image on a scanline by scanline basis which uses a hierarchy of interactive objects and each object includes output information, said render engine analysing said output information of said interactive objects from the lowest to highest interactive objects and determining the output required of each object whereby each interactive object outputs the required number of scanlines necessary as input for higher rank objects to cause the upper most object to output a scanline, each interactive object passing the required number of scanlines as input for the next highest active object and repeating the process until the highest ranked active object produces the particular scanline of the image, and repeating the process for the next scanline of the image until the entire image has been rendered.

18. A method of defining an image comprising

defining a hierarchy of interactive objects, each object including object output requirements necessary for the object to output a scanline input for the next highest interactive object, analysing said interactive hierarchy of objects and determining the required input and output information of said hierarchy of interactive objects for returning a scanline of the image and using said interactive objects from lowest to highest to output the required information for the highest ranked interactive object to output a scanline.

19. A method as claimed in claim 18 including rerendering only a portion of an image affected by a change in one of said interactive objects by analysing said hierarchy of interactive objects to determine the scanlines of the image affected by the change in the objects and rerendering the changed scanlines to update the image.